Parallel Debugging, Prototyping, Data Analysis, and Visualization of Very Large Scale Molecular Dynamics Simulations

David M. Beazley

Theoretical Division (T-11)
Los Alamos National Laboratory
Los Alamos, NM 87545

beazley@lanl.gov

September 12, 1996

Collaborators : P. Lomdahl, S. Zhou, B. Holian, N. Jensen, W. Kerr, T. Germann

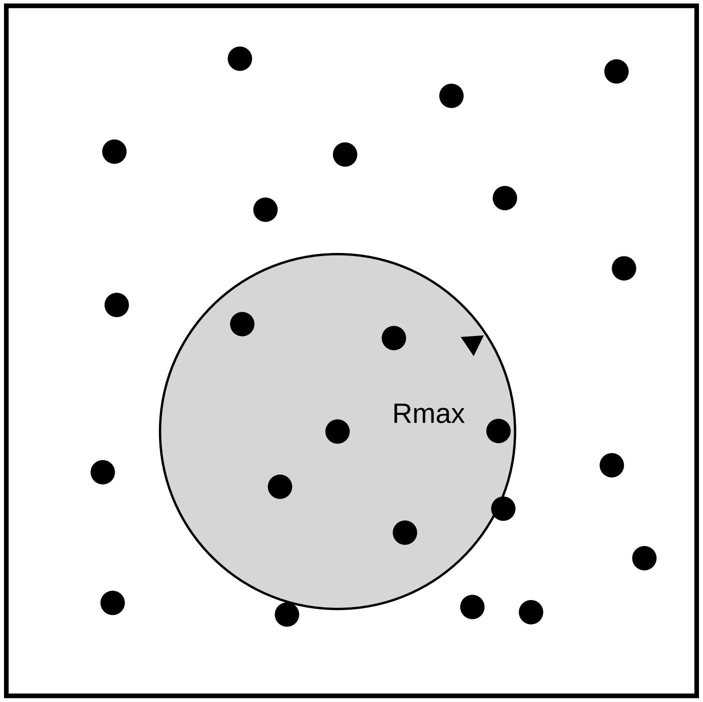
Seven talks in one....

- **The Data Glut**
- **“Computational Steering”**
- **Python**
- **SWIG**
- **The SPaSM code**
- **Parallel debugging and prototyping**
- **Visualization and data analysis**

Disclaimer : This is work in progress!

Short-range Molecular Dynamics

- **Solve F=ma for large collection of N particles**
- **Particles have limited range of interaction.**
- **Physics encoded into potential energy function. (ie. pair-potential, EAM, Stillenger-Weber, etc...).**
- **Basic algorithm :**
  - **Figure out which particles interact**
  - **Calculate forces**
  - **Move particles**
  - **Repeat**

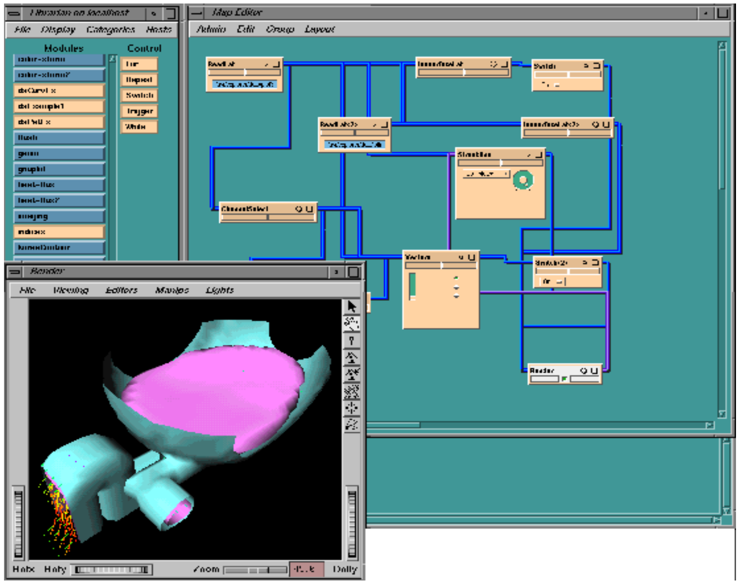


Too Much Data!

- **Developed the SPaSM code in 1992, simulated 130 million atoms in tests.**
- **Production run with 104 million atoms (1994)**
  - **Ran for 180 hours on 512 processor CM-5.**
  - **Generated 40 files, 1.6 Gbytes each**
  - **Would have produced more files if we had a bigger disk quota.**
- **Typical simulations generate tens of Gbytes of data.**
  (would like to produce hundreds of gigabytes)
- **Problems :**
  - **Where do you put the data?**
  - **How do you analyze it?**
  - **What do you analyze it with?**
  - **How do you get it there?**
- **Note : ORNL ran a test simulation with 1 billion atoms in 1994. Yikes!**

Welcome to Tool Hell

Just hook the file gizmo to the particle widget, then tweak the thingabob with slider. Meanwhile, we'll just make 10 quick copies of your dataset before crashing.



- **What planet are these people on?**
- **Performance is horrible:**

  1000 frame animation of 20 Mbyte datafiles (1 million atoms)
  - **3 hours simulation time on 128 processor CM-5**
  - **12 days visualization on dedicated SGI Onyx and Crimson**
- **Require expensive graphics workstations**
  - **All SPaSM users have low-end Sun,HP, and SGI machines**
- **Problem solving environments ARE THE PROBLEM!**
  - **Too complicated, too hard to modify, and nearly useless for doing real work (as far as we can tell).**

Availability of Resources

We'll just buy a really big graphics workstation (or a CAVE), a high speed network, and all of our problems will be solved.

- **Simulations will always grow to use available machine resources.**
- **Why would a sane person think a workstation can handle data generated by 1000 workstations?**
- **Requiring special purpose visualization machines inherently hinders progress (by limiting access).**
- **Just how much interactivity is *really* needed?**
- **If you can't work from your office, then what's the point?**

The Real Problem : Methodology

- **Traditional Scientific Computing**
  - Spend a few days with a small problem to pick parameters
  - Scale it up to a big simulation
  - Run for 10's of hours and keep your fingers crossed.
  - Try to figure out the resulting datafiles.
- **Decoupled simulation and data analysis.**
  - Incomplete data in dump files (unless you're wealthy)
  - Limited flexibility of analysis tools
  - User is detached from the simulation
  - Complicates debugging considerably
- **Other issues**
  - Users are writing code and running simulations
  - Goal is science, not computer science.
  - Huge monolithic systems are counter-productive (and doomed)
  - Need to figure out ways to make working with large simulations as simple as working with small simulations.

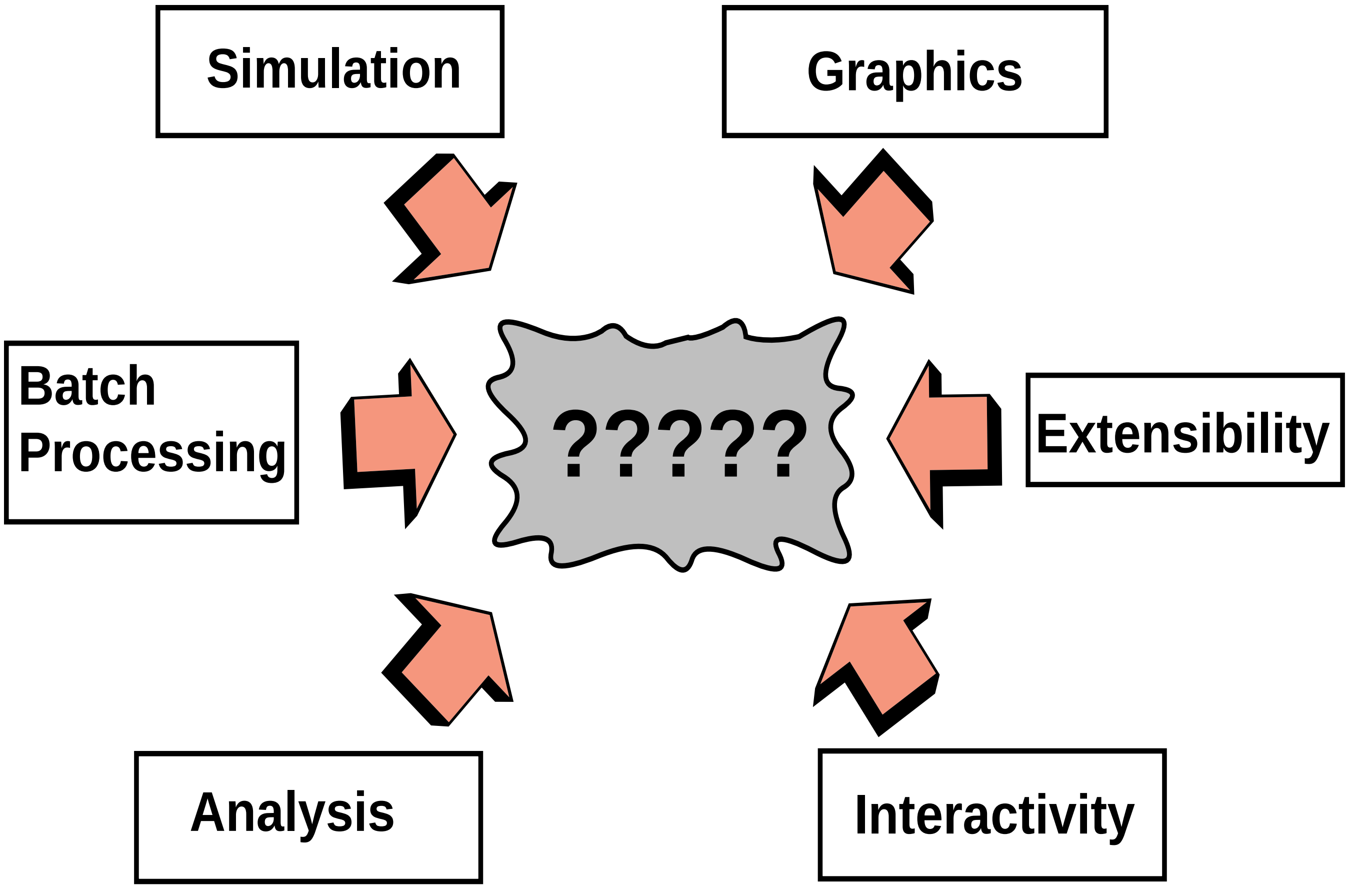
"Computational Steering"

- **The latest buzzword? Perhaps...**
- **Basic Idea (in a nutshell) :**

  Combine simulation, visualization, and data analysis, into an interactive system.

- **Breaks cycle of batch processing**
  (but doesn't replace it)
- **Eliminates tool problems (by eliminating the tools)**
- **Code is reused in a variety of applications**
    - **Simulation**
    - **Visualization**
    - **Data Analysis**
- **The hard part - how do you do it?**

The Problem



How do you combine all of this stuff without creating the code from hell?

Our Approach

- **Add a scripting language interface to our code**
  - **Like Mathematica, MATLAB, IDL, Maple, etc...**
- **Build reusable components**
- **Why this is a good idea :**
  - **Conceptually simple.**
  - **Supports batch processing (scripts).**
  - **Memory efficient.**
  - **Can be easily integrated with C/C++ code.**
  - **Can write code extensions in the interface language itself.**
  - **Highly portable.**
  - **Doesn’t require expensive hardware.**
  - **Can be used over low-bandwidth networks.**
  - **Can even write a GUI (using Tk for instance)**

The Python Language

- **What is it?**

  An exceptionally clean, easy to learn, object oriented scripting language that's, well, a real programming language and fun to use.

- **Why "Python?"**

  Named after Monty Python's Flying Circus

- **Who created it?**

  Guido van Rossum, Stichting Mathematisch Centrum, Amsterdam.

- **Influences**

  Modula-3, ABC, Smalltalk, Lisp, and the UNIX shell

- **I've never heard of it, where is it used?**
  - **Lawrence Livermore National Laboratory**
  - **NASA**
  - **WWW Applications (InfoSeek, Nat'l Geographic, CRNI)**
  - **Red Hat**
  - **Many others...**

Python Features

- Uses a small core and is fully object oriented
- Easily extended with new datatypes and modules
- Dynamically typed
- Very clean and easy to understand syntax
- Portable (runs on UNIX, Win32, Mac)
- Easily extended with C/C++ code.
- Large number of modules :
  - OpenGL
  - Oracle, Sybase
  - UNIX libraries
  - Text processing (regular expressions, etc...)
  - Image processing library
  - Numerical Extension
  - ILU,CORBA

A Brief Taste of Python

Creating variables:

```
a = 3.4*5
b = 2*a - Dt
```

Functions:

```
def fact(n):
    if n <=1:
        return 1
    else :
        return n*fact(n-1)

a = fact(10)
```

Lists:

```
l = [4,5.5,9,10]
l.append(20)
a = l[2:4]
m = [l,a]
```

Loops:

```
for i in range(0,100):
    print i
while i < 100:
    a[i] = 0.0
    i = i + 1
```

Classes:

```
class Complex:
    def __init__(self,re,im):
        self.real = re
        self.imag = im
    def re(self):
        return self.real
    def im(self):
        return self.imag
    def __add__(a,b):
        re = a.real + b.real
        im = a.imag + b.imag
        return Complex(re,im)
```

Exceptions:

```
raise IOError
...
try:
    print 1.0/x
except ZeroDivisionError:
    print '*** has no inverse'
```

Modules:

```
import regex
regex.match("[0-9]+",a)
```

Parallelizing Python

- **The big problem : I/O (surprise)**
  - **What happens when Python tries to do I/O on a parallel machine? (Doing nothing about it won't work.)**
- **Parallel I/O wrapper library**

```
/* File : pstio.h */
#define fprintf  PIO_fprintf
#define fread    PIO_fread
... etc ...
```

- **Implement the I/O wrappers using message passing**
  - **About 1000 lines of C code**
- **Put the following in the Python header file**

```
#include “pstdio.h”
#include  <stdio.h>
```

- **Amazingly, this works!**

Accessing C Functions

- **A C function :**

```
int fcc_block(double Xmin,double Ymin,double Zmin,
              double Xmax,double Ymax,double Zmax, double density);
```

- **A Python Wrapper Function :**

```
PyObject *wrap_fcc_block(PyObject *self, PyObject *args) {
    int result;
    double arg0,arg1,arg2,arg3,arg4,arg5,arg6;
    if (!PyArg_ParseTuple(args,"fffffff",&arg0,&arg1,&arg2,&arg3,&arg4,
                          &arg5,&arg6)
        return NULL;
    result = fcc_block(arg0,arg1,arg2,arg3, arg4,arg5,arg6);
    return Py_BuildValue("d",result);
}
```

- **Python designed to be extended**

- **But, quickly gets tedious.**

Which brings us to....

SWIG

- **Simplified Wrapper and Interface Generator**

- **Automatically generates wrapper code from C/C++**
  - **Uses ANSI C/C++ syntax**
  - **Creates functions, global variables, and constants**
  - **Supports all C built-in types**
  - **Pointers to classes, structures, arrays, etc...**
  - **Supports Python, Tcl, Perl5, Perl4, and Guile-iii**
  - **C++ classes**
  - **Automatically generates documentation**
  - **Supports multiple files and modules**
  - **Easy to use**

- **Is free and fully documented**

A Simple Example

fact.c

```
int fact(int n) {
   if (n <= 1) return 1;
   else return(n*fact(n-1));
}
```

fact.i (SWIG Interface File)

```
%module fact
%{
/* put header files here */
%}
extern int fact(int);
```

SWIG

```
unix> swig -python fact.i
unix> gcc -c fact.c fact_wrap.c -I/usr/local/include/Py
unix> ld -shared fact.o fact_wrap.o -o factmodule.so
unix% python1.3
Python1.3 (Apr 12 1996)   [GCC 2.5.8]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from example import *
>>> n = fact(6);
>>> print n
720
>>>
```

A More Complex Example

Interface file :

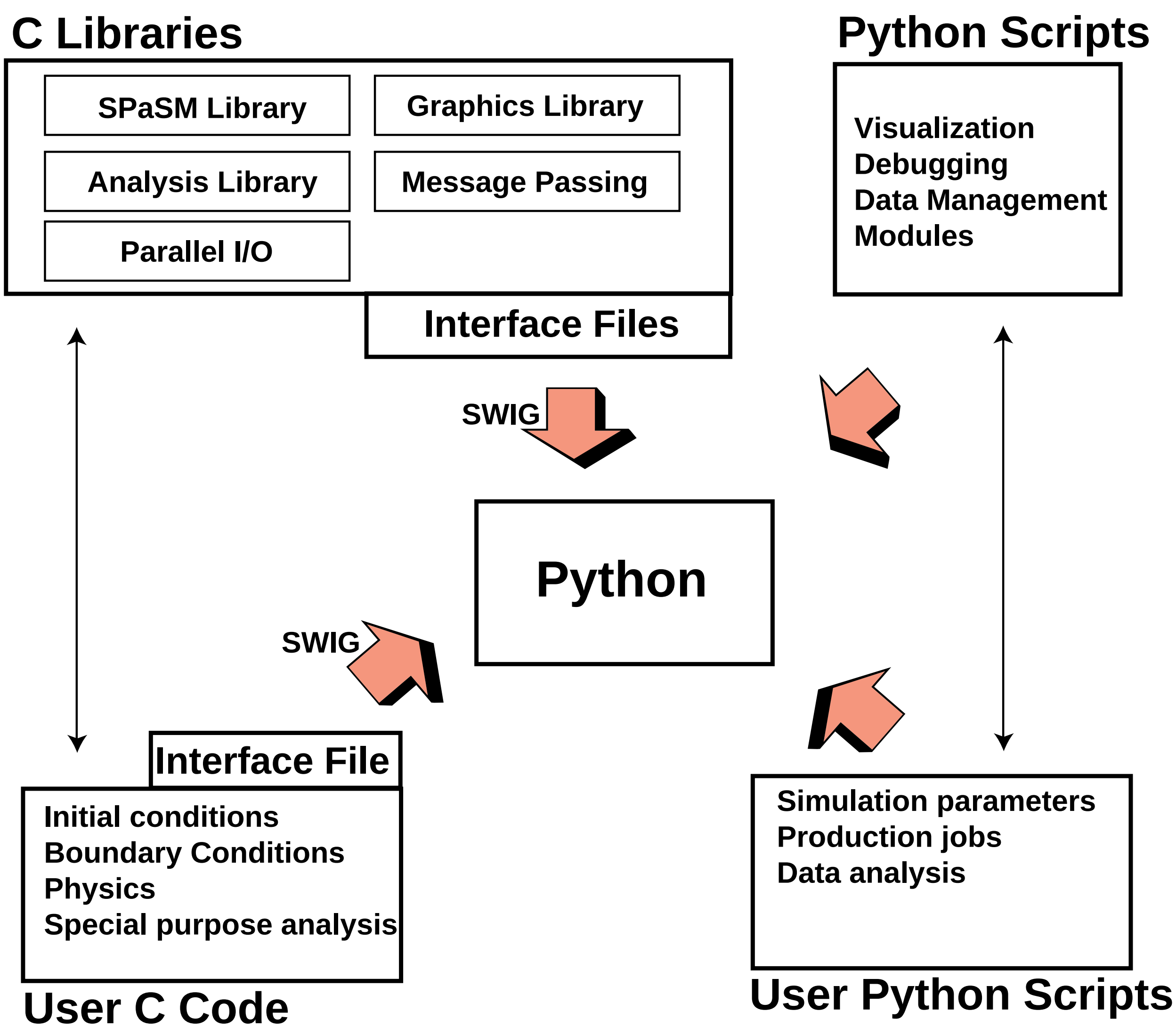
```
%module particle
%{
#include "SPaSM.h"
%}
typedef struct {
    double x,y,z;
} Vector;

extern Vector *new_Vector();

typedef struct {
    Vector r,v,f;
    int     type;
} Particle;

extern Particle *new_Particle();
```

Python :

```
>>> v = new_Vector()
>>> v.x = 3.4
>>> v.y = -2.3
>>> v.z = 0.0
>>> p = new_Particle()
>>> p.v = v
>>> p.r.x = 0.0
>>> p.r.y = 1.0
>>> p.r.z = 0.5
>>> p.type = 1
>>> print p.v.x,p.v.y,p.v.z
3.4 -2.3 0.0
>>>
```

SWIG provides direct access to complex datatypes, pointers and other C/C++ constructs.

Putting it all together



(there will be a quiz later...)

How it works

```
// crack.i.   Interface file for crack problem.
%module crack
%{
#include “crack.h”
%}
%include SPaSM.i
%include analysis.i

extern int ic_crack(int nx, int ny, int nz, double lc);
extern void init_lj(double epsilon,double sigma, double cutoff);
extern void set_boundary_periodic(void);
extern void set_boundary_free(void);
extern void set_boundary_expand(void);
extern void apply_strain(double ex, double ey, double ez);
extern void set_initial_strain(double ex, double ey,double ez);
extern void set_strainrate(double exdot0, double eydot0, double ezdot0);
```

```
SPaSM 3.0 (alpha) ==== Run 182 on cm5-5 ==== Mon Sep  9 19:49:51 1996

Using Python 1.3 (Sep  8 1996)  [GCC 2.6.3]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

SPaSM [182] > SPaSM_geometry(0,0,0,50,50,50,2.5)
SPaSM [182] > init_lj(1.0,1.0,2.5)
SPaSM [182] > set_boundary_periodic()
SPaSM [182] > ic_crack(100,100,20,20.0)
```

A Typical Simulation Script

```
# Simple shock wave problem

nx             = 15
ny             = 15
nz             = 300
shock_velocity = 8.5
temp           = 0.1
width          = 0.3333
r0             = 1.0901733      # Lattice spacing
gap            = 0.10           # Gap (% of z length)
cutoff         = 2.0            # Interaction cutoff
cvar.Dt        = 0.0025         # Timestep

set_path("/sda/sda1/beazley/shock2")

# Only set up initial condition first time
if cvar.Restart == 0:
    ic_shock(nx,ny,nz,shock_velocity,width,gap,temp,r0,cutoff)

# Run it
init_lj(1,1,cutoff)
set_boundary_periodic()
timesteps(10000,25,25,500)
```

Parallel Debugging

- **Can access variables, execute functions, and examine data interactively from Python.**

- **Parallel Python supports debugging on multiple processors**

```
SPaSM [183] > p = first_particle()
SPaSM [183] > print p
r : x = 0.870869, y = 0.100000, z = 46.252136
v : x = -0.219844, y = 0.166902, z = 8.619538

SPaSM [183] > pn(64)
(pn 64) SPaSM [183] > print p
r : x = 0.100000, y = 0.100000, z = 278.283685
v : x = -0.379652, y = 0.434523, z = 0.374103

(pn 64) SPaSM [183] > pnset([4,5,20])
[4, 5, 20] SPaSM [183] > pprint(p)
[4] r : x = 0.870869, y = 6.266951, z = 46.252136
v : x = 0.603058, y = 0.267254, z = 8.635593

[5] r : x = 7.037820, y = 6.266951, z = 46.252136
v : x = 0.060840, y = -0.556352, z = 8.685358

[20] r : x = 0.870869, y = 6.266951, z = 69.378204
v : x = 0.092104, y = 0.362281, z = 8.583000
```

- **Basically, an application specific debugger...**

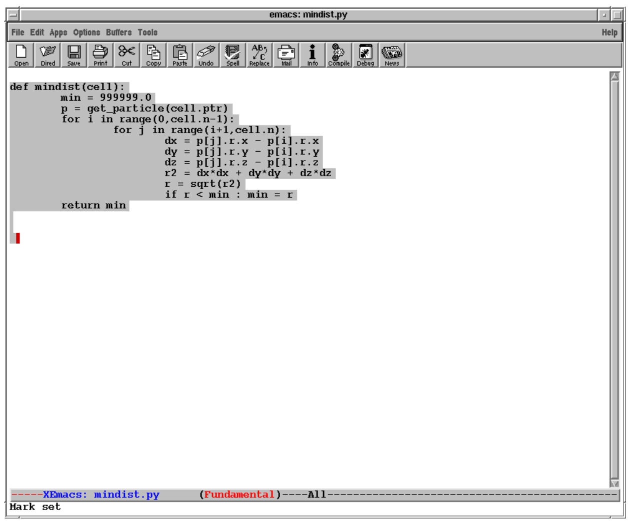
Advanced Debugging

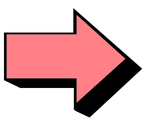
- **Can be difficult to check certain things.   Write a script for it.**

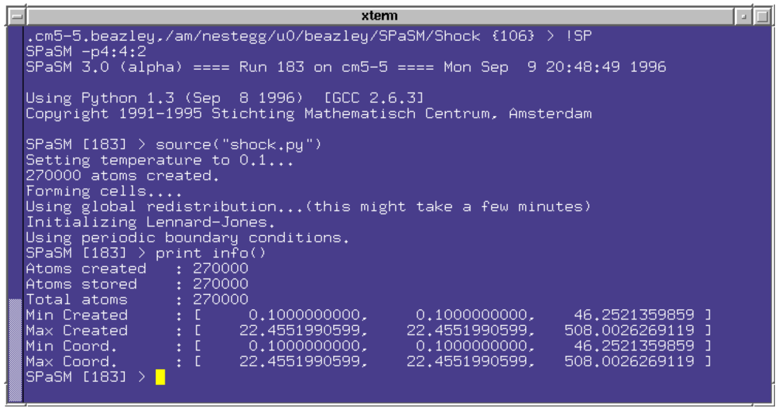
```
def mindist(cell):
    min = 999999.0
    p = get_particle(cell.ptr)
    for i in range(0,cell.n-1):
        for j in range(i+1,cell.n):
           dx = p[j].r.x - p[i].r.x
           dy = p[j].r.y - p[i].r.y
           dz = p[j].r.z - p[i].r.z
           r2 = dx*dx + dy*dy + dz*dz
           r = sqrt(r2)
           if r < min : min = r
    return min
```

```
SPaSM [183] > c = get_cell(1,1,50)
SPaSM [183] > print c
Cell [ ptr = 2d6600, n = 24 ]
SPaSM [183] > source("mindist.py")
SPaSM [183] > mindist(c)
1.0901733
SPaSM [183] >
```

- **Can even cut and paste from emacs--repeatedly!**







Prototyping

- Can write new initial conditions, analysis, and diagnostic code entirely in Python
- No need to recompile SPaSM
- Code runs slower, but development time is much faster.
- Great for one-time procedures, special purpose operations, etc...
- Easy to reimplement in C later.

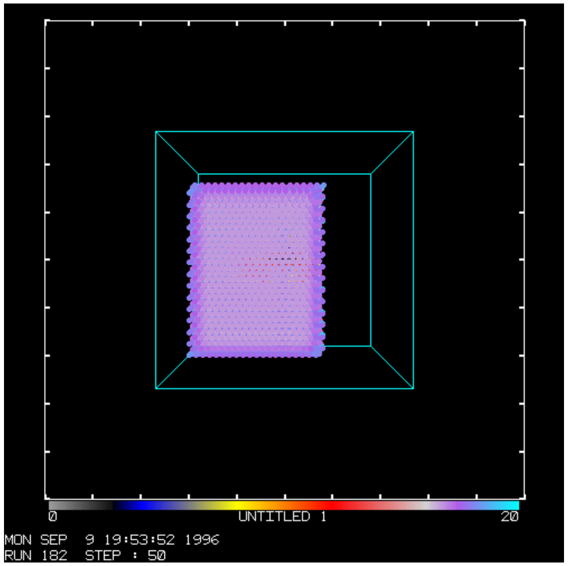
Parallel Visualization

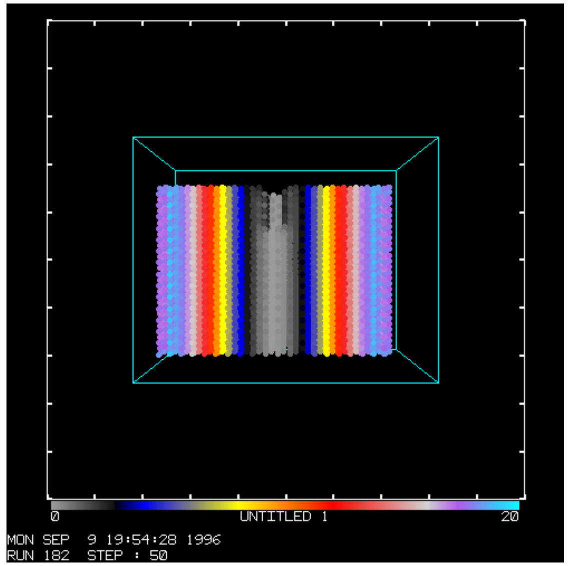
- **Have developed an 8-bit graphics library**
  - **Memory efficiency**
  - **Performance**
- **Features**
  - **2D and 3D graphics primitives**
  - **zbuffering**
  - **Polygons and Spheres**
  - **Creates GIF files as output**
- **Graphics module is stand-alone code.**
  - **Can be compiled independently of the SPaSM code.**
- **Consists of about 5000 lines of ANSI C**
  - **Provides only graphics primitives, nothing else**
  - **About 90 functions at last count...**
- **But a graphics library alone doesn't make a visualization system...**

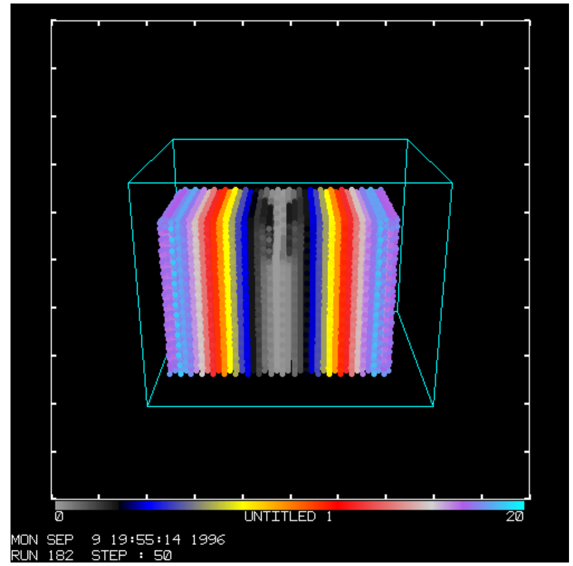
Parallel Data Analysis

- Have developed an object-oriented data analysis system
  - Written almost entirely in Python with C used for high performance
  - Built on top of the graphics library
- Components
  - Collection of Python Base Classes (1100 lines).
  - Collection of different image types (classes). (700 lines)
  - C functions for really high performance stuff. (2000 lines)
    - Extracting particle data
    - Making Spheres
    - Looping over all the particles
  - All of the tedious stuff work is done in Python
    - Tick marks
    - Graph annotation
    - Frames
    - Colorbars
- Easily Customized (in Python)
- Image display via UNIX sockets and xv

Basic Visualization







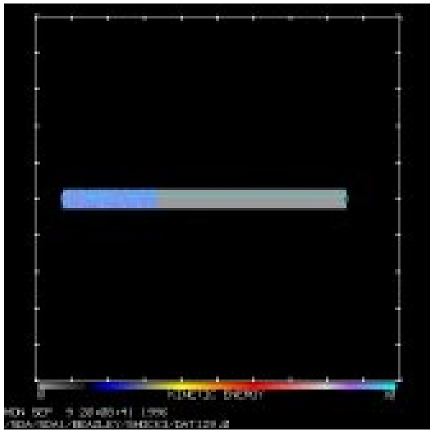

```
ke=Spheres(KE,0,20)
```

```
ke.rotr(90)
ke.show()
```

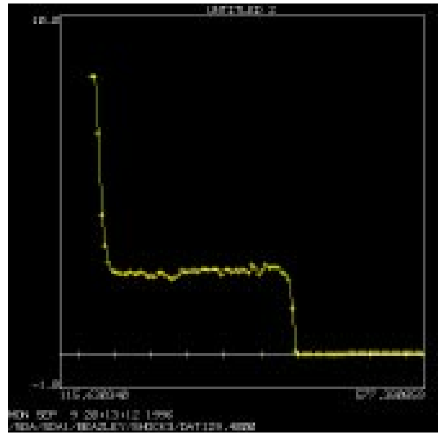
```
rotd(20)
```

- **Images are objects**
- **Can perform various operations on images**
  rotation, translation, zoom, change color ranges, etc...
- **Can create images at any time**

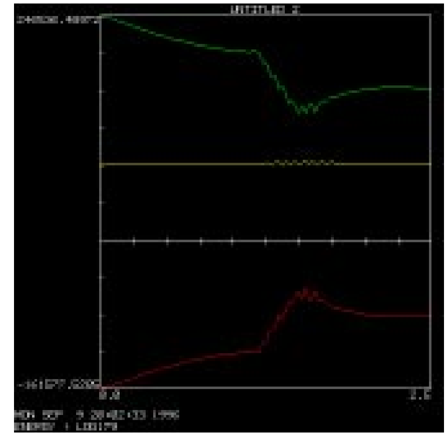
Types of Images



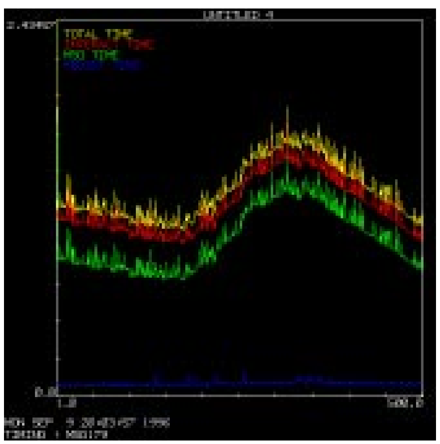
`Spheres(KE,0,10)`



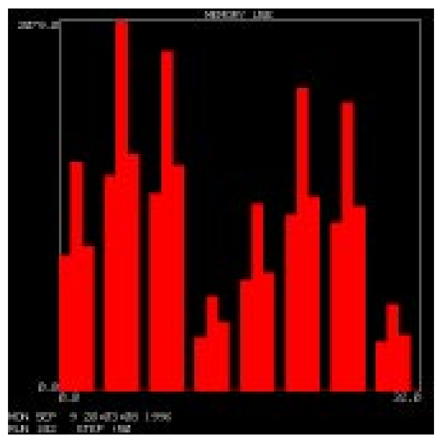
`Profilez(VZ,100)`



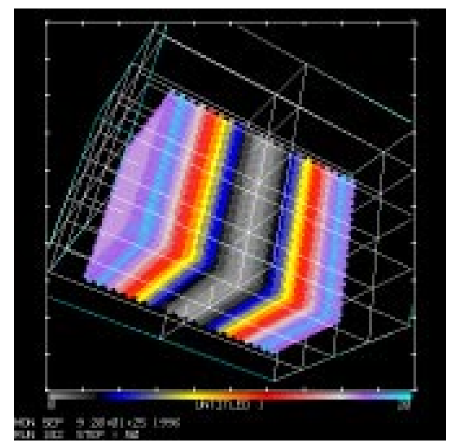
`Energy()`



`Performance()`

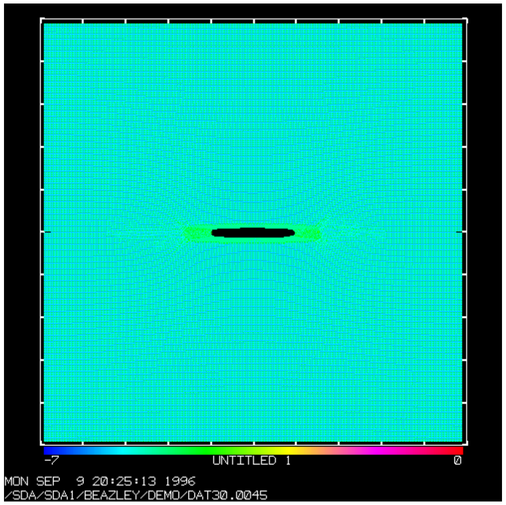


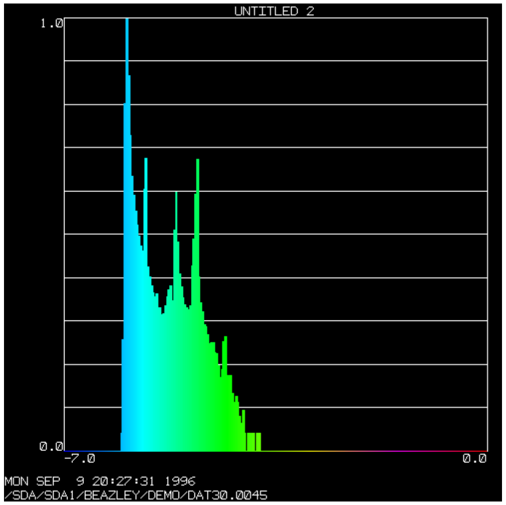
`MemoryUse()`

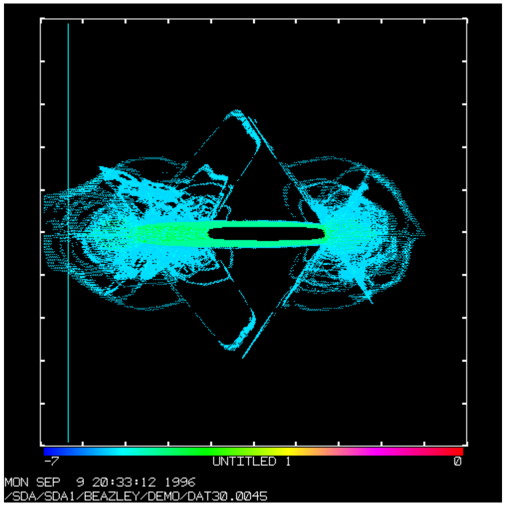


`show_processors()`

Feature Extraction

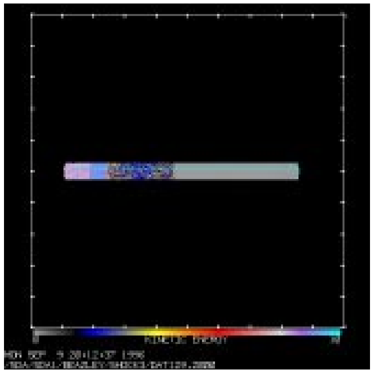




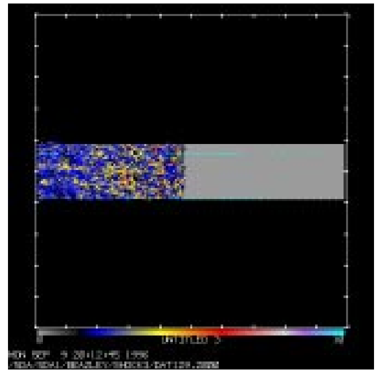


- **Images from 35 Million atom Cu experiment**
  - **< 15 seconds to generate and display on 16 processor T3D**
- **Feature extraction is often difficult**
- **Can look at data distribution (histograms)**
- **Culling and clipping of datasets supported.**

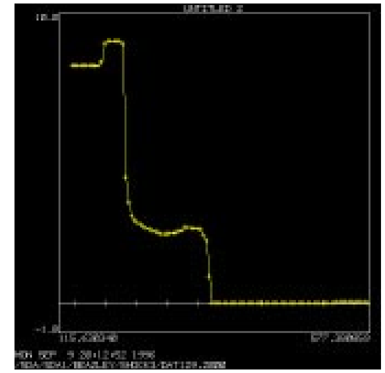
Movie Making



ke



ke2



vz

- **Can make movies from arbitrary groups of images**

```
make_movie("Dat129.0",25,1000,ke,ke2,vz)
```

- **Produce multiple movies simultaneously**
  - **Pick multiple views, make one pass through the data**
- **Can often produce >1000 images in an hour.**
- **Runs as a batch process---high performance.**

Odds and Ends

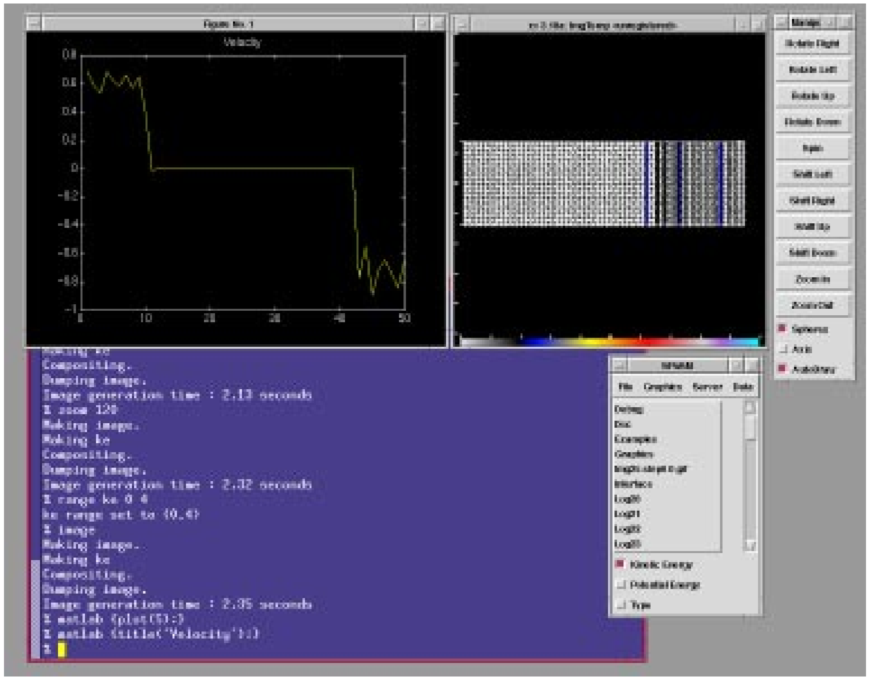
- **Dynamic loading of modules**

  On workstations, and SMPs, it's possible for Python to dynamically load C code right into a running simulation, providing even more modularity.

- **Language Independence**

  While we primarily use Python, SPaSM can be compiled to use Tcl or Perl interfaces without modification

- **SPaSM Molecular Dynamics Code**
- **MATLAB C API**
- **Data Analysis**
- **Tcl/Tk GUI**



Results

We have been developing this system for over a year

- **Analysis of arbitrarily large datasets possible from any UNIX workstation and over internet connections.**
- **Has revolutionized the way we use the code.**
- **Modularity has increased the reliability of the code and resulted in a decrease in code size!**
- **Current system only incurs a 10% memory overhead over old code.**
- **No noticeable impact on performance (most of the work is still done in C anyways).**
- **High-powered data analysis, without having to use a commercial package or rewriting everything in C++.**
- **Much work remains...**

Limitations

- **There are no limitations**

- **Actually, almost all of this has been an experiment born of frustration.**

- **Code is radically different than old version**
  - **Big monolithic systems are on their way out. Code forces one to think in terms of independent modules and packages.**

  - **Often difficult to know how various modules interact and which order to perform various operations (even though it may not matter).**

- **Graphics quality has been sacraficed for performance and memory efficiency**
  - **I feel that this is acceptable for real work.**

Future Directions

- **Continued development of the system**
- **Shift towards shared memory multiprocessors and workstation clusters.**
- **More sophisticated analysis techniques**
- **An OpenGL graphics back-end (interchangable with the current graphics library).**
- **Release of our lightweight graphics library and parallel Python.**
- **Long term : Scientific Databases**

  Dealing with large amounts of data is not a visualization problem, but a data management problem. Need innovative techniques for managing and organizing datasets, images, results, etc...

Resources

The Python homepage :

`http://www.python.org`

The SWIG homepage :

`http://www.cs.utah.edu/~beazley/SWIG`